CS406 – Parallel Computing
FUTOSHIKI HW1

Oguzhan Ilter
19584
February 2020

1 Introduction

The aim for this assignment is to design and implement an algorithm that can solve efficiently (in terms of run-time and memory consumption) any given solvable Futoshiki Puzzle. In the implementation, there should be no parallelization but SIMD operations, prefetching for the sake of providing spatial/temporal locality and preprocessing are allowed and encouraged.

In Futoshiki puzzles, the objective of the game is to discover the digits hidden inside the board's cells; each cell is filled with a digit between 1 and the board's size. On each row and column each digit appears exactly once. Some digits may be given at the start. Inequality constraints are initially specified between some of the squares, such that one must be higher or lower than its neighbor. These constraints must be satisfied in order to complete the puzzle.

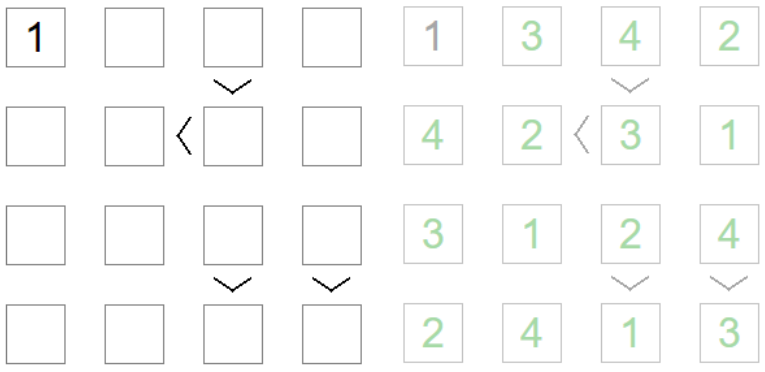


Figure 1: Example of Futoshiki Puzzle and its answer

In the following sections, the implementations of different approaches along with their advantage and disadvantage are going to be discussed. In addition, execution time of the algorithms will be provided in a table.

2 Implementation

I have implemented five different algorithms that use same structure to represent grids and grid map. The algorithms differ from each other in terms of puzzle solving approaches and preprocessing types.

The implementation is a combination of logical rules. Values of each node restrict the possible number domain of other nodes according to rules that are explained in the previous section. However, in any case, it is a recursive algorithm and relies on the fact, values that are previously assigned to other grids are correct.

At every iteration (recursion call), possible values domain of related grids (*biggerThan*, *smallerThan*, *column*, *row*) are updated according to current (*focus*) grid value. If a possible number domain set of a node becomes empty due to the value of *focus* grid, the recursion *returns false* so that *gridMap* is recalled to last valid state and the recursion run on from that state.

In my experiments I noticed that it is very crucial how the *focus* grid has been selected at every recursion. On one hand selection procedures require preprocessing which increases execution time, on the other hand preprocessing decrease the total execution time at the end.

Algorithm General Structure of Recursive Solver
Require (*matrix, constraints, size*)

```
1: IF emptyGridCount == 0 then
2:          return true
3: ELSE
4:          →Select focus_grid
5           FOR ALL possible values of focus_grid
6:                   focus_grid.Value ← a value from domain set
7:
8:                   IF no empty domain then
9:                            IF solver() then                >> recursive call
10:                                    return false
11:                  go back to last valid state
12:         →return false
```

The following run-time table shows the execution time of each algorithm with different inputs in microseconds. The values for deterministic algorithms (non-random algorithms) are observed when the cache is hot, but for the random algorithms these values are mean of 50 to 500 iterations.

Inputs / Algorithms	input1	input1_2	input2	input2_2	input3
first_available	250	101	1161	11715	12255
empty_grid_first	269	103	1215	12147	12606
empty_grid_random	268	119	3717	10454	549417
min_remaining_domain	49	51	148	124	286
min_remaining_domain_random	89	70	135	227	1108

Inputs / Algorithms	input3_2	input4	input4_2	input5	input5_2
first_available	89698	2137941	1689264	55265949	- -
empty_grid_first	93072	2211360	1728955	57061734	- -
empty_grid_random	6894596	- -	- -	- -	- -
min_remaining_domain	15224	9168	237148	2598498	28048120
min_remaining_domain_random	6990	7671	178295	2452317	5969259

Table 1: Hot Cache, 50 -500 times run, microseconds

first_available method is the brute force method. It checks every grid at every recursion call to find a suitable grid to make changes (in this case it searches for an empty grid). Its performance is better (about %25 faster) in hot cache when the input grid map size is smaller than four however the difference between hot and cold cache situations becomes negligible as the grid map size gests bigger. The mean cache hits ratio for all inputs is around %78.

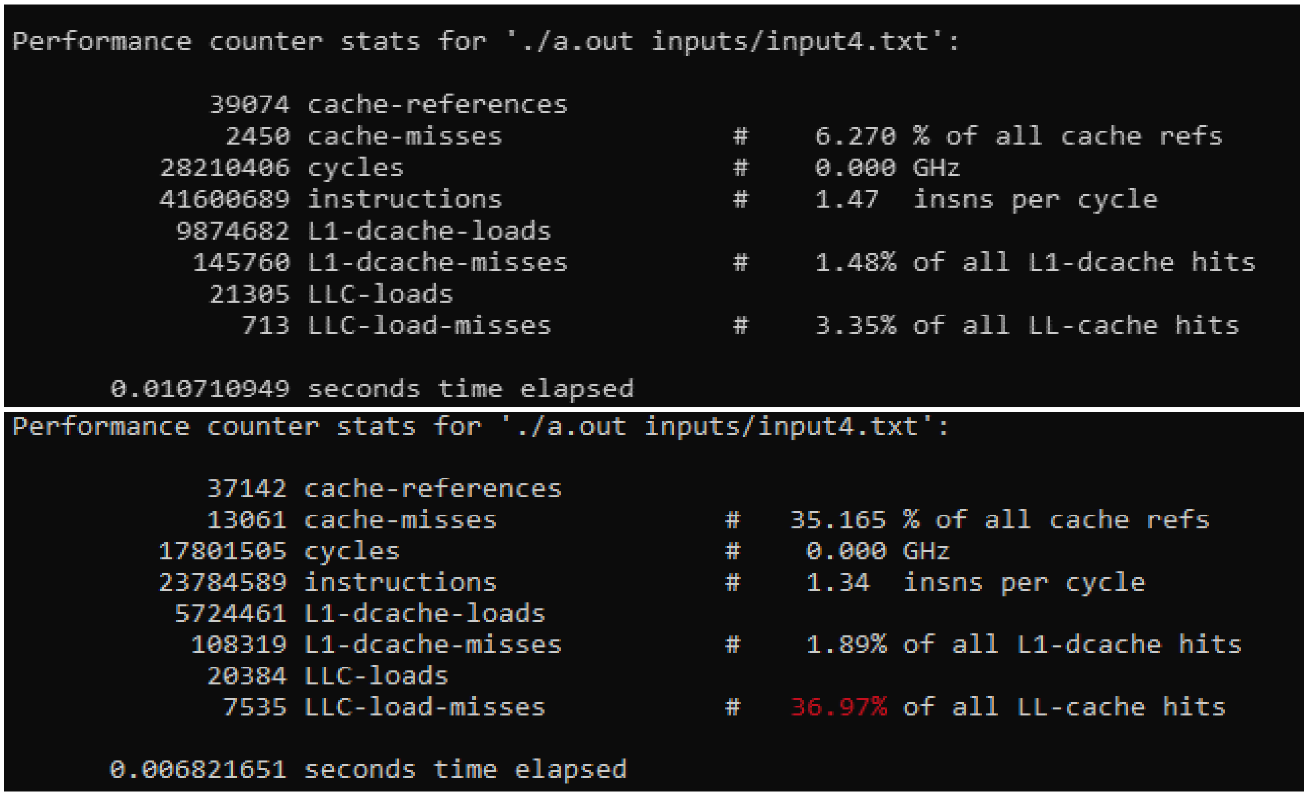
empty_grid_first method keeps the set of empty grids and pick the first grid in the set at every recursive call. In this method, preprocessing does not bring any good to final execution time. In fact, it makes %1 to %2 slower the algorithm. The performance difference between hot and cold cache situation is the same as previous method. The mean cache hits ratio for all inputs is around %60.

empty_grid_first_random method selects randomly an empty grid at every recursive call. This method does not bring any progress. Since it is very slow in many cases I did not examine this method further.

The remaining two methods will be discussed and compared in the following section, since there is a huge gap with previous methods in terms of execution time.

3 Comparison and Explanation of Best Algorithms

The difference between these two algorithms and previous ones, is the *focus* grid selection algorithm. It is interesting enough that I added one more preprocessing function on top of the *empty_grids_first* method and the performance of final algorithm became much better than any previous algorithm. The cash miss ratio has been dropped significantly, since the preprocessing functions reduce the number of possible iterations (recursive calls) and eliminate the unnecessary fetch operations because it iterates over adjacent grids on both *arrangeDomains()* and *solve()*. This fact increases spatial locality need of the algorithm. The miss ratio is between 4% to &40 for both algorithms.



```
Performance counter stats for './a.out inputs/input4.txt':

            39074 cache-references
             2450 cache-misses              #    6.270 % of all cache refs
         28210406 cycles                    #    0.000 GHz
         41600689 instructions              #    1.47  insns per cycle
          9874682 L1-dcache-loads
           145760 L1-dcache-misses          #    1.48% of all L1-dcache hits
            21305 LLC-loads
              713 LLC-load-misses           #    3.35% of all LL-cache hits

      0.010710949 seconds time elapsed
```

```
Performance counter stats for './a.out inputs/input4.txt':

            37142 cache-references
            13061 cache-misses              #   35.165 % of all cache refs
         17801505 cycles                    #    0.000 GHz
         23784589 instructions              #    1.34  insns per cycle
          5724461 L1-dcache-loads
           108319 L1-dcache-misses          #    1.89% of all L1-dcache hits
            20384 LLC-loads
             7535 LLC-load-misses           #   36.97% of all LL-cache hits

      0.006821651 seconds time elapsed
```

Figure 2: Cache Hits ratio

These two algorithms, namely *min_remaining_domain* and *min_remaining_domain_random*, differ from each other in the way that they pick the *focus* grid. As the name implies *min_remaining_domain_random* picks randomly a grid from the set of grids that have the minimum domain size among the empty grids.

The comparison with two algorithms is especially hard since they are better at different situations. When the size of grid map (puzzle) is small, *min_remaining_domain* gives better results on the other hand, *min_remaining_domain_random* runs faster in larger grid maps. In addition, *min_remaining_domain_random's* best result is significantly better.

Inputs / timing	input1	input1_2	input2	input2_2	input3	input3_2	input4	input4_2	input5	input5_2
min	59	59	112	118	228	4661	2460	347	974046	1486227
max	156	135	196	566	3788	9505	15582	869802	5388896	10083553
mean	88	70	135	276	1115	6978	7741	178461	2658743	5480037

Table 2: min, max and runtime of random algorithm (time in microsecond)

As it is shown in the table 2, min values are very fast. In the following table (table 3) we can see the hot cache and cold cache run time of *min_remaining_domain* method.

Inputs / cache state	input1	input1_2	input2	input2_2	input3	input3_2	input4	input4_2	input5	input5_2
hot cache	49	51	148	124	286	15224	9168	237632	2598498	28048120
cold cache	91	92	210	194	403	15342	9286	237632	2592822	28070642

Table 3: hot cache and cold cache runtime of deterministic algorithm (time in microsecond)

When we compare hot and cold cache situations the gap closes after the input3_2. However, when we compare with *min_remaining_domain_random,* its best cases are significantly faster.

Unfortunately, I could not implement any prefetching mechanism since any of the array size is not constant therefore in some steps, prefetching might have caused unnecessary processing delays which cause increase in overall runtime.

As a result, I picked *min_remaining_domain_random* method as my main method. If we run all this examples one by one the total time that we gain by using *min_remaining_domain_random* method in worst case is over 14.5 second. The reason why I picked this algorithm has been explained further in comments section. Please check last section.

4 Compile and Run

I compiled the codes via *gcc 8.2.0* with commend on terminal *g++ -std=c++11 xxxx.cpp -g -O3*.

In addition, I tested their cache miss ratios:

```
perf stat -B -e cache-references,cache-misses,cycles,instructions,L1-dcache-loads,L1-dcache-misses,LLC-loads,LLC-load-misses ./a.out inputs/inputx.txt
```

5 Comments

The randomized algorithm has the possibility to pick the grid that has the biggest effect on other grids. By using this logic, I will improve further my deterministic code by adding one more layer in preprocessing. This preprocessing layer will look ahead and give to every grid a *power_constant* which represent that the number of will be changed grids when that specific grid has been changed.

6 Last Minute Change!

I managed to implement the algorithm that is described in comments section. The algorithm uses one more preprocessing layer to pick *focus* grid. It uses the same structures that are used in *min_remaining_domain_random*. However, after it obtains the vector that contains minimum domain number grids, it searches the grid that would have maximum effect on other grids, when that specific grid would have been updated. In addition, this new algorithm is deterministic and gives same results in every iteration. Therefore, I will use *min_remaining_max_effect* method as my main method.

input1	input1_2	input2	input2_2	input3	input3_2	input4	input4_2	input5	input5_2
108	108	202	274	780	5812	9928	30354	1600875	3218268

Table 4: new algorithm's runtime microseconds

In order to enhance the speed of the algorithm, it is needed to implement a careful prefetching mechanism. Because the algorithm gets many cache miss.